Programmation C et Système
Introduction au langage C à partir de Java

Régis Barbanchon

L2 Informatique

Exemple de prog en Java affichant des mentions de notes

```java
// File: Grade.java
import java.lang.System; // useless as java.lang.* always imported

public class Grade {
  public static void printGrade (int grade) {
    if      (grade < 10) System.out.printf ("%2d: X\n" , grade);
    else if (grade < 12) System.out.printf ("%2d: P\n" , grade);
    else if (grade < 14) System.out.printf ("%2d: AB\n", grade);
    else if (grade < 16) System.out.printf ("%2d: B\n" , grade);
    else                 System.out.printf ("%2d: TB\n", grade);
  }

  public static void printAllGrades (int minGrade, int maxGrade) {
    for (int k= minGrade; k <= maxGrade; k++) {
      printGrade (k);
    }
  }

  public static void main (String[] args) {
    int minGrade= 11, maxGrade= 14;
    printAllGrades (minGrade, maxGrade);
    System.exit (0); // useless as it is the default behavior
  }
}
```

Compilons `Grade.java` en `Grade.class` et exécutons-le :

```
$ javac Grade.java
$ java Grade
11: P
12: AB
13: AB
14: B
```

Le même exemple traduit en C

```c
// File: Grade.c
#include <stdlib.h> // exit()
#include <stdio.h> // fprintf, stdout

void printGrade (int grade) {
  if      (grade < 10) fprintf (stdout, "%2d: X\n" , grade);
  else if (grade < 12) fprintf (stdout, "%2d: P\n" , grade);
  else if (grade < 14) fprintf (stdout, "%2d: AB\n", grade);
  else if (grade < 16) fprintf (stdout, "%2d: B\n" , grade);
  else                 fprintf (stdout, "%2d: TB\n", grade);
}

void printAllGrades (int minGrade, int maxGrade) {
 for (int k= minGrade; k <= maxGrade; k++) {
   printGrade (k);
 }
}

int main (void) { // we don't use the command-line arguments here
  int minGrade= 11, maxGrade= 14;
  printAllGrades (minGrade, maxGrade);
  return 0; // or from any function: exit(0);
}
```

Compilons `Grade.c` en un exécutable `Grade`, et exécutons-le :

```
$ clang -std=c99 -W -Wall -pedantic Grade.c -o Grade
$ ./Grade
11: P
12: AB
13: AB
14: B
```

Points communs et différences Java et C

- Utiliser un module : `#include` en C, `import` en Java.
- Coder une routine : fonction en C, méthode de classe en Java.
- même contrôle de flot de base dans les deux langages :

```
{ doThis; ...; doThat; }
```

```
if (someConditionHolds) doThis; else doThat;
```

```
switch (intExpr) { case val: doThis; break; ...; default: doThat; break; }
```

```
while (someConditionHolds) doThis;
```

```
do doThis; while (someConditionHolds);
```

```
for (initialization; someConditionHolds; postIteration) doThis;
```

```
break; // exit from loop
```

```
continue; // skip current loop current iteration
```

```
return;       // exit from function without value (void function)
return value; // exit from function with value
```

Les opérateurs communs au Java et au C

Opérateurs arithmétiques et d'affectation :

```
(expr)
numExpr1 + numExpr2
numExpr1 - numExpr2
numExpr1 * numExpr2
numExpr1 / numExpr2
intExpr1 % intExpr2
- numExpr
+ numExpr
```

```
var     =  expr
numVar  += numExpr
numVar  -= numExpr
numVar  *= numExpr
numVar  /= numExpr
intVar  %= intExpr
```

```
++ numVar
-- numVar
numVar ++
numVar --
```

Opérateur conditionnel, virgule, cast, crochets de tableau :

```
boolExpr ? expr1 : expr2
```

```
expr1, ..., exprN
```

```
(type) expr
```

```
array [intExpr]
```

Opérateurs logiques, de comparaison, d'égalité :

```
boolExpr1 && boolExpr2
boolExpr1 || boolExpr2
! boolExpr
```

```
numExpr1 <  numExpr2
numExpr1 >  numExpr2
numExpr1 <= numExpr2
numExpr1 >= numExpr2
```

```
expr1 == expr2
expr1 != expr2
```

Opérateurs bit-à-bit :

```
intExpr1 &  intExpr2
intExpr1 |  intExpr2
intExpr1 ^  intExpr2
intExpr1 >> intExpr2
intExpr1 << intExpr2
~ expr
```

```
intVar &=  intExpr
intVar |=  intExpr
intVar ^=  intExpr
intVar >>= intExpr
intvar <<= intExpr
```

Les types numériques entiers courants en Java et C

Java possède les types entiers signés suivants :

- `short` : valeurs sur 16 bits dans $-2^{15} \cdots 2^{15} - 1$,
- `int` : valeurs sur 32 bits dans $-2^{31} \cdots 2^{31} - 1$,
- `long` : valeurs sur 64 bits dans $-2^{63} \cdots 2^{63} - 1$.

C possède les mêmes types qui ont les mêmes plages de valeurs *en pratique* sur une architecture 32 ou 64-bits en complément à 2.

C possède en plus les types non-signés correspondants, avec *en pratique* sur une architecture 32 ou 64-bits en complément à 2 :

- `unsigned short` : valeurs sur 16 bits dans $0 \cdots 2^{16} - 1$,
- `unsigned int` : valeurs sur 32 bits dans $0 \cdots 2^{32} - 1$,
- `unsigned long` : valeurs sur 64 bits dans $0 \cdots 2^{64} - 1$.

La norme ISO définissant le langage C est plus complexe et définit seulement des garanties minimales de plages dont certaines sont dépassées en pratique depuis 30 ans.

Les types numériques flottants en Java et C

Java possède les types flottants suivants :

- `float:` simple précision sur 32 bits,
- `double:` double précision sur 64 bits.

On retrouve ces même types en C, en pratique au format IEEE 754.
On utilise quasiment toujours `double` par défaut.

En C, on peut assigner une valeur à une variable type moins précis sans utilisation d'un cast explicite vers le type d'arrivée :

```
double d= 5.5;
float  f= d; // legal in C, illegal in Java
int    i= f; // legal in C, illegal in Java
```

En Java, ce n'est pas possible, le typage étant plus strict,
et un cast explicite vers le type d'arrivée est nécessaire :

```
double d= 5.5;
float  f= (float) d; // cast from double to float
int    i= (int)   f; // cast from float  to int
```

Remarque : La syntaxe du cast en C et en Java est la même.

Les Booléens en Java et en C

En Java, le type `boolean` n'est pas un type numérique.
Les valeurs `false` et `true` de Java ne sont pas des nombres.

En C, toute valeur numérique nulle ou référence `NULL` signifie *faux*.
Et toute valeur numérique ou référence non-nulle signifie *vrai*.
Les fonctions de la librairie standard utilisent généralement `int`
pour retourner un Booléen, et *vrai* n'est pas toujours normalisé à `1`.

Par exemple, le prédicat `int isupper(char c)` de `<ctype.h>`
retourne un entier non-nul lorsque le caractère `c` est une majuscule
mais ce n'est généralement pas une valeur normalisée à `1`.

Depuis C99, le header `<stdbool.h>` définit le type entier `bool`
qui n'a que deux valeurs entières, `0` et `1`, ainsi que les macros :

```
#define false 0
#define true  1
```

Toute valeur non-nulle stockée dans un `bool` se transforme en `1`.

Le type `char` diffère en Java et C

En Java, on a les deux types entiers suivants :

- `byte` : valeurs signées sur 8 bits dans $-2^7 \cdots 2^7 - 1$,
- `char` : valeurs non-signées sur 16 bits dans $0 \cdots 2^{16} - 1$.

En C, on a sur une architecture 32 ou 64-bits en complément à 2 :

- `signed char` : vals signées sur 8 bits dans $-2^7 \cdots 2^7 - 1$,
- `unsigned char` : vals non-signées sur 8 bits dans $0 \cdots 2^8 - 1$,
- `char` : l'un des deux ci-dessus, au choix du compilateur.

En C, l'incertitude sur la *signedness* de `char` n'est pas sans poser problème, et il faut donc parfois le caster en `unsigned char`.

En pratique, pour les plages de valeurs sur ces architectures :

- `char` en Java $\equiv$ `unsigned short` ou `wchar_t` en C,
- `byte` en Java $\equiv$ `signed char` en C.

Les constantes littérales en Java et C

Même conventions pour les constantes littérales en Java et en C :

- entiers par défaut en en **base 10** : `-123`, `999`, ...
- préfixe `0` pour un entier en **base 8** : `0123`, `0777`, ...
- préfixe `0x` pour un entier en **base 16** : `0x12B`, `0xFFF`, ...
- suffixe `L` pour un **long** : `999999999L`, ...
- point décimal pour un **double** : `3.1415`, ...
- suffixe `f` pour un **float** : `3.1415f`, ...
- notation `e` pour $\times$ **10^n** : `9.5e3`, ...
- quotes `''` pour un **caractère** : `'a'`, `'\n'`, `'\''`, `'\\'`, ...
- guillemets `""` pour une **chaîne** : `"Regis"`, `"lol\n"`, ...

Uniquement en Java, pas en C :

- préfixe `0b` pour un entier en **base 2** : `0b1011001`, ...
- undercore `_` comme **séparateur de chiffres** : `1_200_500`, ...

Uniquement en C, pas en Java :

- suffixe `U` pour un **unsigned** : `255U` , ...

Les chaînes en Java et C

En Java, les chaînes sont des instances de la classe `String`.
En C, les chaînes sont des tableaux de `char`, terminés par `'\0'`.
On parle de *null-terminated strings* :

```
char name1[]= "Regis";
char name2[]= { 'R', 'e', 'g', 'i', 's', '\0' };
```

Ici, les tableaux `name1` et `name2` sont identiques.
De telles chaînes ne connaissent pas leur longueurs,
autrement qu'en recherchant l'index du caractère nul `'\0'`.
C'est ce que fait la fonction `strlen()` de `<string.h>` :

```
// File: RegisLength.c
#include <string.h> // strlen()
#include <stdio.h>  // fprintf(), stdout

int main (void) {
  int length= strlen ("Regis");
  fprintf (stdout, "length: %d\n", length);
  return 0;
}
```

```
$ clang -std=c99 -W -Wall -pedantic RegisLength.c -o RegisLength
$ ./RegisLength
length: 5
```

Méthodes statiques en Java ≡ fonctions en C

En Java, toute méthode est définie à l'intérieur d'une classe.
Les fonctions C sont l'équivalent des méthodes statiques en Java :

```
class MyMath {
  public static double  doubleSquared (double x) { return x * x; }
  public static int     intSquared    (int n)    { return n * n; }
  public static boolean intIsEven     (int n)    { return n % 2 == 0; }
}
```

En C, comme il n'y a pas de notion de classes ou d'interfaces,
Les fonctions sont donc définies à l'extérieur de toute autre entité :

```
double MyMath_doubleSquared (double x) { return x * x; }
int    MyMath_intSquared    (int n)    { return n * n; }
bool   MyMath_intIsEven     (int n)    { return n % 2 == 0; }
```

Une fonction C sans paramètre se déclare avec `(void)`, et non `()`.
En revanche, elle s'invoque normalement avec `()` :

```
#include <stdio.h>

void sayHello (void) { fprintf (stdout, "Hello!\n"); }

int main (void) {
  sayHello ();
  return 0;
}
```

Pas de surcharge de nom de fonction en C

En Java, on peut avoir deux méthodes de même nom
dans une classe si les paramètres diffèrent en type ou en nombre :

```
class MyMath {
  public static double squared (double x) { return x * x; }
  public static int    squared (int n)    { return n * n; } // method overload
}
```

Il s'agit de la surcharge de méthode (*overload*).
En C, c'est impossible, la fonction est identifiée par son nom seul :

```
double MyMath_squared (double x) { return x * x; }
int    MyMath_squared (int n)    { return n * n; } // error: function redefined
```

En C, chaque fonction doit donc avoir un nom différent des autres :

```
double MyMath_doubleSquared (double x) { return x * x; }
int    MyMath_intSquared    (int n)    { return n * n; }
```

Ordre de définition des fonctions en Java et en C

En Java, l'ordre de définition des méthodes n'est pas important.
Une méthode utilisée peut être définie plus bas dans le fichier :

```
class MyMath {
  public static boolean intIsOdd  (int n) { return ! intIsEven (n); }
  public static boolean intIsEven (int n) { return intHasDivisor (n, 2); }
  public static boolean intHasDivisor (int n, int d) { return n % d == 0; }
}
```

En C, avant qu'une fonction soit utilisée, elle doit être
– soit déjà définie plus haut (donc avec son corps) :

```
bool MyMath_intHasDivisor (int n, int d) { return n % d == 0; }
bool MyMath_intIsEven (int n) { return MyMath_intHasDivisor (n, 2); }
bool MyMath_intIsOdd  (int n) { return ! MyMath_intIsEven (n); }
```

– soit déjà déclarée plus haut (avec son prototype seul) :

```
bool MyMath_intHasDivisor (int n, int d); // declaration
bool MyMath_intIsEven (int n);            // declaration
bool MyMath_intIsOdd  (int n);            // declaration

bool MyMath_intIsOdd  (int n) { return ! MyMath_intIsEven (n); }
bool MyMath_intIsEven (int n) { return MyMath_intHasDivisor (n, 2); }
bool MyMath_intHasDivisor (int n, int d) { return n % d == 0; }
```

Fichiers d'entête `.h` (headers)

En général on regroupe ces déclarations de prototypes dans un fichier d'entête (ou *header*) ayant pour extension `.h` :

```
// File: MyMath.h

#ifndef MY_MATH_H // preprocessing guard against multiple inclusion
#define MY_MATH_H

bool MyMath_intHasDivisor (int n, int d); // declaration
bool MyMath_intIsEven (int n);            // declaration
bool MyMath_intIsOdd  (int n);            // declaration

#endif
```

Et on inclut ce header en tête du fichier `.c` d'implémentation avec la directive `#include` qui copie-colle son contenu sur place :

```
// File:  MyMath.c

#include "MyMath.h"

bool MyMath_intIsOdd  (int n) { return ! MyMath_intIsEven (n); }
bool MyMath_intIsEven (int n) { return MyMath_intHasDivisor (n, 2); }
bool MyMath_intHasDivisor (int n, int d) { return n % d == 0; }
```

On utilise les guillemets `""` pour inclure un header utilisateur.
Les chevrons `<>` sont reservés aux headers standards.

La fonction `main()` en Java

En Java, s'il y a `n` arguments sur la ligne de commande,
ils occupent les indices `0` à `n-1` dans `args[]` de `main()`,
et le nombre d'arguments `args.length == n` :

```
// File: App.java

public class App {
  public static void main (String[] args) {
    System.out.printf ("args.length: %d\n", args.length);
    for (int k= 0; k < args.length; k++) {
      System.out.printf ("args[%d]: <%s>\n", k, args[k]);
    }
    System.exit (args.length > 0 ? 0 : 1);
  }
}
```

```
$ javac App.java # compile to App.class
```

```
$ java App Hi Mr Regis
args.length: 3
args[0]: <Hi>
args[1]: <Mr>
args[2]: <Regis>
```

```
$ echo $? # exit status, 0 for success, 1 to 255 for failure
0
```

La fonction `main()` en C

En C, s'il y a `n` arguments sur la ligne de commande,
ils occupent les indices `1` à `n` dans `argv[]` de `main()`,
`argc == n+1`, et `argv[0]` est le nom de la commande :

```
// File: App.c
#include <stdlib.h> // exit()
#include <stdio.h>  // stdout, fprintf()

int main (int argc, char ** argv) {
  fprintf (stdout, "argc: %d \n", argc);
  for (int k= 0; k < argc; k++) {
    fprintf (stdout, "argv[%d]: <%s>\n", k, argv[k]);
  }
  exit (argc > 1 ? 0 : 1);
}
```

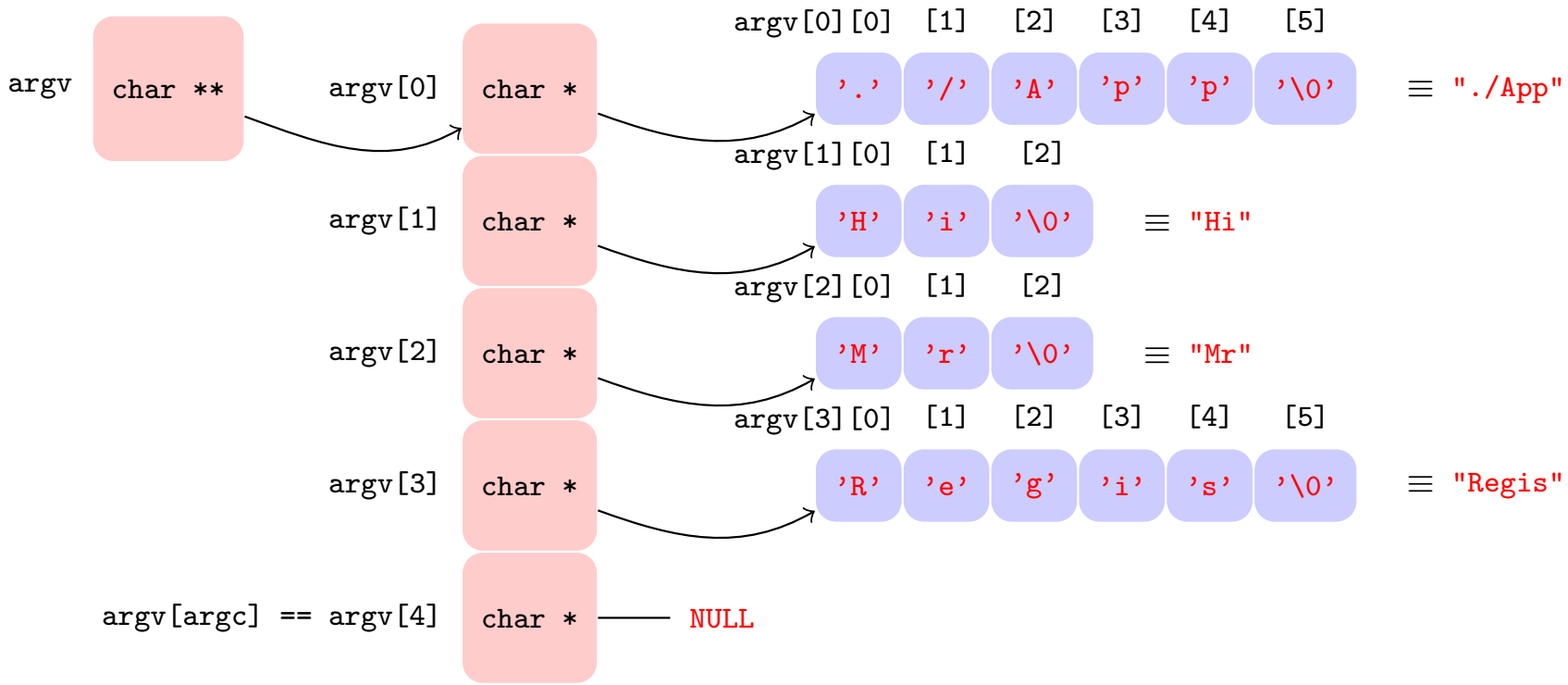
```
$ clang -std=c99 -W -Wall -pedantic App.c -o App
```

```
$ ./App Hi Mr Regis
argc: 4
argv[0]: <./App>
argv[1]: <Hi>
argv[2]: <Mr>
argv[3]: <Regis>
```

```
$ echo $? # exit status, 0 for success, 1 to 255 for failure
0
```

Les arguments `int argc` et `char ** argv`

```
$ ./App Hi Mr Regis
```



`TypeName *` dénote le type *pointeur sur* `TypeName`.
Chaque `argv[k]` pointe sur le premier `char` de l'arg numéro `k`.
Donc chaque `argv[k]` est de type `char *`.
De plus, `argv` pointe sur `argv[0]` qui est de type `char *`.
Donc `argv` est de type `char **`.